如何使用 AVR-GCC

安装 GNU C for AVR

一．执行安装程序

安装程序可以从www.avrfreaks.net下载。

二．生成链接用的库文件

$(AVR)表示安装的根目录。（在本人系统里为 f:\avrgcc）

安装程序并没有生成编译所需的库文件。生成库文件关键是要运行位于$(AVR)下的 RUN.BAT。原程序如下：

```
   @echo off
   if NOT %AVR%!==! goto install
   rem set environment variables
   set AVR=f:\AVRGCC
   set CC=avr-gcc
   set PATH=.;f:\AVRGCC\bin;%path%
   doskey
:install
   if %1!==! GOTO end
   rem install libc
   cd f:\AVRGCC\lib\avr-libc-20010701\src
   rem first win32_make_dirs will make some errors(I don't know why?)
   make -f makefile-win32 win32_make_dirs
```

```
    make -f makefile-win32
    make -f makefile-win32 install
    make -f makefile-win32 clean
:end
    f:
    cd f:\AVRGCC
    mode con: lines=43
```

要修改为：

```
    @echo off
    if NOT %AVR%!==! goto install
    rem set environment variables
    set AVR=f:\AVRGCC
    set CC=avr-gcc
rem    set PATH=.;f:\AVRGCC\bin;%path%
    doskey
:install
rem    if %1!==! GOTO end
    rem install libc
    cd f:\AVRGCC\lib\avr-libc-20010701\src
    rem first win32_make_dirs will make some errors(I don't know why?)
    f:\AVRGCC\bin\make -f makefile-win32 win32_make_dirs
    f:\AVRGCC\bin\make -f makefile-win32
    f:\AVRGCC\bin\make -f makefile-win32 install
```

```
    f:\AVRGCC\bin\make -f makefile-win32 clean
:end
    f:
    cd f:\AVRGCC
    mode con: lines=43
```

在以后的应用中，运行的是修改之前的 RUN.BAT，但要去掉 rem if %1!==! GOTO end 的“rem”。去掉“rem”之后，后续的语句将被跳过。因此 MAKE 部分的“f:\AVRGCC\bin\”可加可不加。实际上，由于我的系统 PATH 太多，造成“set PATH=.;f:\AVRGCC\bin;%path%”运行有问题，而且还有多个 make，造成了歧义，否则，“f:\AVRGCC\bin\”根本就可以不用加，“set PATH=.;f:\AVRGCC\bin;%path%”也不用注释掉。

仔细查看 makefile-win32，其主要工作是编译 Start-Up 模块和各种库文件。而 Start-Up 模块又主要是完成设置中断向量、设置全局变量、清除数据段等工作。用户可以通过修改此文件来适应未来器件。如果用户想添加库文件的话，也可以类似处理：添加新代码（C 或汇编），重新编译。

有新器件出现时需要为其创建必要的文件：

1. 合适的.h 文件：参照各个.h 文件。如果是 TINY 器件，参照 iotnxx.h；如果是一般器件，参照 ioxx.h；如果是 MEGA 器件，参照 iomxx.h。
2. 在 io-avr.h 里添加合适的语句，以便编译器调用正确的头文件。
3. 如果有新的中断向量，则需要在 sig-avr.h 里声明，然后在 gcrt1.s 里用 VEC 语句创建新的向量（注意次序）。

编译和链接应用程序

原文作者为 Rich Neswold(rneswold@enteract.com)。

首先在www.avrfreaks.net上下载测试程序集 gcctest.zip，然后安装。

1. 将 GCCTEST\INCLUDE 下的 MAKE1、MAKE2 拷贝到$(AVR)\ INCLUDE
2. 将工作目录的 MAKEFILE (每个工程都要有一个此文件，且可由自己进行修改以适合自己的应用。如果要利用原有文件，则注意只能有一个 C 文件)中的 MCU、TRG、SRC、ASRC、INC、LIB 等项填入合适的内容
3. 在工作目录运行位于$(AVR)\BIN 下的 MAKE.EXE（注意：由于系统可能存在其他应用程序的 MAKE，因此可能还需要加路径。也可以将其改名。）
4. 从 MAKE1、MAKE2 和 MAKEFILE 可以看出，用户可以修改诸如输出文件名等多种选项。

在 C 代码中嵌入汇编指令

一．GCC 的 ASM 声明

首先看一个从 PORTD 读入数据的例子：

asm(“in %0, %1” : “=r”(value) : “I”(PORTD) :);

由上可以看出嵌入汇编的 4 个部分：

1．汇编指令本身，以字符串“in %0, %1”表示；

2．由逗号分隔的输出操作数，本例为“=r”(value)

3．由逗号分隔的输入操作数，本例为“I”(PORTD)

4．Clobber 寄存器

嵌入汇编的通用格式为：

asm(code : output operand list : input operand list : clobber list);

例子中%0 表示第一个操作数，%1 表示第二个操作数。即：

%0 → “=r”(value)

%1 → “I”(PORTD)

如果在后续的 C 代码中没有使用到汇编代码中使用的变量，则优化编译时会将这些语句删除。为了防止这种情况的发生，需要加入 volatile 属性：

asm volatile (“in %0, %1” : “=r”(value) : “I”(PORTD) :);

嵌入汇编的的 Clobber 寄存器部分可以忽略，而其他部分不能忽略，但可以为空。如下例：

asm volatile(“cli” : :);

二．汇编代码

用户可以在 C 代码里嵌入任意的汇编指令，就如同在汇编器里写程序一样。

AVR-GCC 提供了一些特殊的寄存器名称：

符号	寄存器
__SREG__	状态寄存器 SREG（0x3F）
__SP_H__	堆栈指针高字节（0x3E）
__SP_L__	堆栈指针低字节（0x3D）
__tmp_reg__	r0
__zero_reg__	r1。对于 C 代码而言其值永远为 0

三．输入/输出操作数

约束符号	适用于	范围
a		r16~r23
b	指针	Y，Z
d		r16~r31
e	指针	X，Y，Z
G	浮点常数	0.0
I	6 比特正常数	0~63
J	6 比特负常数	-63~0
l		r0~r15

M	8 比特正常数	0~255
N	整数常数	-1
O	整数常数	8，16，24
P	整数常数	1
r		r0~r31
t		R0
W	寄存器对	r24，r26，r28，r30
X	指针 X	r27:r26
Y	指针 Y	r29:r28
Z	指针 Z	r31:r30

要注意的是，在使用这些约束符号时要防止选择错误。例如，用户选择了”r”约束符号，而汇编语句则使用了”ori”。编译器可以在 r0~r31 之间任意选择寄存器。若选择了 r2~r15，则会由于不适用 ori 而出现编译错误。此时正确的约束符应该是”d”。

约束符号还可以有前置修饰符，如下表所示。

修饰符	**指定**
=	只写操作数
+	读-写操作数（嵌入汇编不支持）
&	寄存器只能用做输出

输出操作数必须为只写操作数，C 表达式结果必须为 1（r0~r15）。编译器不检查汇编指令中的变量类型是否合适。

输入操作数为只读。如果输入/输出使用同一个寄存器怎么办呢？此时可以在输入操作数的约束字符里使用一个一位数字来达到这个目的。这个数字告诉编译器使用与第 n 个（从 0 开始计数）操作数相同的寄存器。例如：

asm volatile(“SWAP %0” : “=r”(value) : “0”(value));

这条语句的目的是交换变量 value 的高低 4 位。约束符号“0”告诉编译器使用与第一个操作数相同的寄存器作为输入寄存器。要注意的是，即使用户没有指定，编译器也有可能使用相同的寄存器作为输入/输出。在某些情况下会引发严重的问题。如果用户需要区分输入/输出寄存器，则必须为输出操作数增加修饰

符”&”。如下例所示。

```
asm volatile(“in %0, %1；
                out %1, %2”
              : “=&r”(input)
              : “I”(port), “r”(output)
             );
```

此例的目的是读入端口数据，然后给端口写入另一个数据。若编译器不幸使用了同一个寄存器作为参数 input 和 output 存储位置，则第一条指令执行后 output 的内容就被破坏了。而用了修饰符”&”之后，这个问题得以解决。

下面为一个高 16 位与低 16 位交换的 32 位数据操作的例子：

```
asm volatile(“mov    __tmp_reg__, %A0；
              mov    %A0, %D0；
              mov    %D0, __tmp_reg__；
             mov    __tmp_reg__, %B0；
             mov    %B0, %C0；
             mov    %C0, __tmp_reg__”
             : “=r”(value)
             : “0”(value)
             )
```

“0”代表第一个操作数，“A”，“B”，“C”，“D”表示：

31……24	23……16	15……8	7……0
D	C	B	A

四．Clobber

如前所示，asm 语句的最后一部分为 clobber。如果用户在汇编代码里使用了没

有作为操作数声明的寄存器，就需要在 clobber 里声明以通知编译器。下面为一个中断无关的加一操作例子。

```
asm volatile(“cli;
        ld r24, %a0;
        inc r24;
        st %a0, r24;
        sei”
        :
        :”z”(ptr)
        :”r24”
        )
```

编译结果为：

```
CLI
LD R24， Z
INC R24
ST Z， R24
SEI
```

当然，用户也可以用__tmp_reg__来取代 r24。此时就没有 clobber 寄存器了。

下面为考虑更详细的例子：

```
c_func
{
    uint_t s;
    asm volatile(“in %0, __SREG__;
```

```
        cli;
        ld, __tmp_reg__, %a1;
        inc __tmp_reg__;
        st %a1, __tmp_reg__;
        out __SREG__, %0”
        : “=r”(t)
        :”z”(ptr)
    );
}
```

现在看起来好象没问题了。其实不尽然。由于优化的原因，编译器不会更新 C 代码里其他使用这个数值的寄存器。出于同样的优化原因，上述代码的输入寄存器可能保持的不是当前最新的数值。用户可以加入特殊的”memory” clobber 来强迫编译器及时更新所有的变量。

更好的方法是将一个指针声明为 volatile，如下所示：

```
volatile uint8_t *ptr;
```

这样，一旦指针指向的变量发生变化，编译器就会重新加载最新的数值。

使用 ELFCOFF

使用 AVR-GCC 生成的是.ELF 和.OBJ 文件，可以由 AVRSTUDIO 直接调用。可惜的是无法在调试环境里查看变量。为此 Mr. Flavio Gobber 编写了一个将 ELF 文件转换为 COFF 文件的程序，以此来达到在调试环境里查看变量的目的。这个程序可以在 www.avrfreaks.net 之 AVR-GCC 属性页下载。解压缩后生成 ELFCOFF 目录。然后将\ElfCoff\avrgcc 目录对应 avrgcc 目录，把\ElfCoff\avrgcc

目录下的各个子目录的内容拷贝到相应的 avrgcc 目录下的各个子目录。然后可可以运行了。运行结果即可生成.COFF 文件。

API

嵌入式编程的代码可以简单地分为两部分，一是与硬件无关的算法部分，对其编程与普通 C 编程没有区别；二是与硬件相关的寄存器/端口操作部分。不同的 MCU 实现方法各有不同。在 AVR-GCC 里则通过一系列的 API 来解决。当然，用户也可以定义自己的 API。在此简单地介绍目前 AVR-GCC 里定义的 API，以及 AVR-GCC 的工作过程。

一．应用程序启动过程(Start Up)

标准库文件包含一个启动模块(Start Up Module)，用于为真正执行用户程序做环境设置。

启动模块完成的任务如下：

1. 提供缺省向量表
2. 提供缺省中断程序入口
3. 初始化全局变量
4. 初始化看门狗
5. 初始化寄存器 MCUCR
6. 初始化数据段
7. 将数据段.bss 的内容清零
8. 跳转到 main()。（不用调用方式，因为 main()不用返回）

启动模块包含缺省中断向量表，其内容为预先定义好的函数名称。这些函数名称可以由程序员重载。中断向量表的第一个内容为复位向量，执行结果是将程

序跳转到_init_。在启动模块里，_init_表示的地址与_real_init_指向的地址相同。如果要加入客户代码，则需要在程序里定义一个_init_函数。在此函数的末尾跳转到_real_init_。具体实现如下：

```
void _real_init_(void);
void _init_(void) __attribute__((naked));
void _init_(void)
{
    // 用户代码

    // 最后的代码必须为：
    asm ("rjmp _real_init_");
}
```

在_real_init_部分，系统将设置看门狗和 MCUCR 寄存器。启动模块并没有真正取用相应寄存器的设置数值（以符号_init_wdctr_，_init_mcucr_，_init_emcucr_表示），而是通过地址来取得其值。因而用户可以通过链接器的--defsym 选项来设置这些符号的地址。如果用户没有定义，则启动模块将使用缺省值。

接下来系统将从程序存储器里把具有初值的全局变量加载到数据存储器 SRAM。然后是将数据段.bbs 清零。此数据段包含所有没有的初值的非 AUTO 变量。

最后，系统跳转到 main()函数，用户代码开始执行。系统对此特殊函数加入一些特殊的处理。进入此函数后，堆栈指向 SRAM 的末尾。

二．存储器 API

AVR 具有三种存储器：FLASH，SRAM 和 EEPROM。AVR-GCC 将程序代码放在 FLASH，数据放在 SRAM。

I．程序存储器

如果要将数据（如常量，字符串，等等）放在 FLASH 里，用户需要指明数据类型__attribute__((progmem))。为了方便使用，AVR-GCC 定义了一些更直观的符号，如下表所示。

类型	定义
prog_void	void __attribute__((progmem))
prog_char	char __attribute__((progmem))
prog_int	int __attribute__((progmem))
prog_long	long __attribute__((progmem))
prog_long_long	long long __attribute__((progmem))
PGM_P	prog_char const*
PGM_VOID_P	prog_void const*

提供的库函数有：

1．__elpm_inline

用法：uint8_t __elpm_inline(uint32_t addr);

说明：执行 ELPM 指令从 FLASH 里取数。参数为 32 位地址，返回一个 8 位数据。

2．__lpm_inline

用法：uint8_t __elpm_inline(uint16_t addr);

说明：执行 LPM 指令从 FLASH 里取数。参数为 16 位地址，返回一个 8 位数据。

3．memcpy_P

用法：void* memcpy_P(void* dst, PGM_VOID_P src, size_t n);

说明：memcpy 的特殊版本。完成从 FLASH 取 n 个字节的任务。

4．PRG_RDB

用法：uint8_t PGR_RDB(uint16_t addr);

说明：此函数简单地调用__lpm_inline

5．PSTR

用法：PSTR（s）;

说明：参数为字符串。功能是将其放在 FLASH 里并返回地址。

6．strcmp_P

用法：int strcmp(char const*, PGM_P);

说明：功能与 strcmp()类似。第二个参数指向程序存储器内的字符串。

7．strcpy_P

用法：char* strcpy_P(char*, PGM_P);

说明：功能与 strcpy()类似。第二个参数指向程序存储器内的字符串。

8．strlen_P

用法：size_t strlen_P(PGM_P);

说明：功能与 strlen()类似。第二个参数指向程序存储器内的字符串。

9．strncmp_P

用法：size_t strncmp_P(char const*, PGM_P, size_t);

说明：功能与 strncmp()类似。第二个参数指向程序存储器内的字符串。

10．strncpy_P

用法：size_t strncpy_P(char*, PGM_P, size_t);

说明：功能与 strncpy()类似。第二个参数指向程序存储器内的字符串。

II．EEPROM

AVR 内部有 EEPROM，但地址空间与 SRAM 的不相同。在访问时必须通过 I/O 寄存器来进行。EEPROM API 封装了这些功能，为用户提供了高级接口。使用时要包含 eeprom.h。在程序里定义 EEPROM 数据的例子如下：

static uint8_t variable_x __attribute__((section(".eeprom"))) = 0;

不同的 AVR 器件具有不同数目的 EEPROM。链接器将针对不同的器件分配存储器空间。

1．eeprom_is_ready

用法：int eeprom_is_ready(void);

说明：此函数用于指示是否可以访问 EEPROM。如果 EEPROM 正在执行写操作，则在 4ms 内无法访问。此函数查询相应的状态位来指示现在是否可以访问 EEPROM。

2．eeprom_rb

用法：uint8_t eeprom_rb(uint16_t addr);

说明：从 EEPROM 里读出一个字节的内容。参数 addr 用于指示要读出的地址。_EEGET(addr)调用此函数。

3．Eeprom_read_block

用法：void eeprom_read_block(void* buf, uint16_t addr, size_t n);

说明：读出一块 EEROM 的内容。参数 addr 为起始地址，n 表明要读取的字节数。数据被读到 SRAM 的 buf 里。

4．eeprom_rw

用法：unint16_t eeprom_rw(uint16_t addr);

说明：从 EEPROM 里读出一个 16 位的数据。低字节为低 8 位，高字节为高 8 位。参数 addr 为地址。

5．eeprom_wb

用法：void eeprom_wb(uint16_t addr, uint8_t val);

说明：将 8 位数据 val 写入地址为 addr 的 EEPROM 存储器里。_EEPUT（addr，val）调用此函数。

三．中断 API

由于 C 语言设计目标为硬件无关，因此各种编译器在处理中断时使用的方法都是编译器设计者自己的方法。

在 AVR-GCC 环境里，向量表已经预先定义，并指向具有预定义名称的中断例程。通过使用合适的名称，用户例程就可以由相应的中断所调用。如果用户没有定义自己的中断例程，则器件库的缺省例程被加入。

除了中断向量表的问题，编译器还必须处理相关寄存器保护的问题。中断 API 解决了细节问题。用户只要将中断例程定义为 INTERRUPT（）或 SIGAL（）即可。而对于用户没有定义的中断，缺省例程的处理是 reti 指令。

函数定义可参见 interrupt.h，中断信号符号表参见 sig-avr.h。

1．cli

用法：void cli(void);

说明：通过置位全局中断屏蔽位来禁止中断。其编译结果仅为一条汇编指令。

2．enable_external_int

用法：void enable_external_int(uint8_t ints);

说明：此函数访问 GIMSK 寄存器（对于 MEGA 器件则是 EIMSK 寄存器）。功能与宏 outp()一样。

3．INTERRUPT

用法：INTERRUPT（signame）

说明：定义中断源 signame 对应的中断例程。在执行时，全局屏蔽位将清零，其他中断被使能。ADC 结束中断例程的例子如下所示：

```
INTERRUPT（SIG_ADC）
{
}
```

4．sei

用法：void sei(void);

说明：通过清零全局中断屏蔽位来使能中断。其编译结果仅为一条汇编指令。

5．SIGNAL

用法：SIGNAL（signame）

说明：定义中断源 signame 对应的中断例程。在执行时，全局屏蔽位保持置位，其他中断被禁止。ADC 结束中断例程的例子如下所示：

```
SIGNAL（SIG_ADC）
{
}
```

6．timer_enable_int

用法：void timer_enable_int(uint8_t ints);

说明：此函数操作 TIMSK 寄存器。也可以通过 outp()来设置。

四．I/O API

1．BV

用法：BV(pos);

说明：将位定义转换成屏蔽码(MASK)。与头文件 io.h 里的位定义一起使用。例如，置位 WDTOE 和 WDE 可表示为“**BV(WDTOE) | BV(WDE)**”

2．bit_is_clear

用法：uint8_t bit_is_clear(uint8_t port, uint8_t bit);

描述：如果 port 的 bit 位清零则返回 1。此函数调用 sbic 指令，故 port 应为有效地址。

3．bit_is_set

用法：uint8_t bit_is_set(uint8_t port, uint8_t bit);

描述：如果 port 的 bit 位置位则返回 1。此函数调用 sbis 指令，故 port 应为有效地址。

4．cbi

用法：void cbi(uint8_t port, uint8_t bit);

说明：清零 port 的 bit 位。bit 的值为 0~7。如果 port 为实际 I/O 寄存器，则此函数生成一条 cbi 指令；否则，函数生成相应的优化代码。

5．inp

用法：uint8_t inp(uint8_t port);

说明：从端口 port 读入 8 比特的数值。如果 port 为常数，则函数生成一条 in 指令；若为变量，则函数用直接寻址指令。

6．__inw

用法：uint16_t __inw(uint8_t port);

说明：从 I/O 寄存器读入 16 位的数值。此函数用于读取 16 位寄存器（ADC，ICR1，OCR1，TCNT1）的值，因为读取这些寄存器需要合适的步骤。由于此函数只产生两条汇编指令，因此要在中断禁止时使用，否则有可能由于中断插入到指

令之间造成读取错误。

7．__inw_atomic

用法：uint16_t __inw_atomic(uint8_t port);

说明：以原子语句方式读取 16 位 I/O 寄存器的数值。此函数首先禁止中断，读取数据之后再恢复中断状态，因此可以安全地应用在各种系统状态。

8．loop_until_bit_is_clear

用法：oidoid loop_until_bit_is_clear (uint8_t port, uint8_t bit);

说明：此函数简单地调用 sbic 指令来测试端口 port 的 bit 位是否清零。port 必须为有效端口。

9．loop_until_bit_is_set

用法：oidoid loop_until_bit_is_set (uint8_t port, uint8_t bit);

说明：此函数简单地调用 sbis 指令来测试端口 port 的 bit 位是否置位。port 必须为有效端口。

10．outp

用法：void outp(uint8_t val, uint8_t port);

说明：将 val 写入端口 port。如果 port 为常数，则函数生成一条 out 指令；若为变量，则函数用直接寻址指令。

11．__outw

用法：void __outw(uint16_t val, uint8_t port);

说明：将 16 位的 val 写入端口 port。此函数适合于操作 16 位寄存器，如 ADC，ICR1，OCR1，TCNT1。由于此函数只产生两条汇编指令，因此要在中断禁止时使用，否则有可能由于中断插入到指令之间造成读取错误。

12．__outw_atomic

用法：void __outw_atomic(uint16_t val, uint8_t port);

说明：将 16 位的 val 写入端口 port。此函数适合于操作 16 位寄存器，如 ADC，ICR1，OCR1，TCNT1。此函数首先禁止中断，读取数据之后再恢复中断状态，因此可以安全地应用在各种系统状态。

13．sbi

用法：void sbi(uint8_t port, uint8_t bit);

说明：置位 port 的 bit 位。bit 的值为 0~7。如果 port 为实际 I/O 寄存器，则此函数生成一条 sbi 指令；否则，函数生成相应的优化代码。

五．看门狗 API

以下函数操作看门狗。宏定义参见 wdt.h。

用户可以通过起动代码初始化看门狗。WDTCR 的缺省值为 0。如果你希望将其设置为其他值，则需要在链接命令里加入相应的命令。使用的符号为__init_wdtcr__。如下为将 WDTCR 设置为 0x1f 的例子：

avr-ld –defsym __init_wdtcr__=0x1f

1．wdt_disable

用法：void wdt_disable(void);

说明：关闭看门狗。

2．wdt_enable

用法：void wdt_disable(unit8_t timeout);

说明：使能看门狗。看门狗溢出时间为 timeout。

timeout	**周期**
0	16K CLK
1	32K CLK
2	64K CLK
3	128K CLK
4	256K CLK

5	512K CLK
6	1024K CLK
7	2048K CLK

3．wdt_reset

用法：void wdt_reset(void);

说明：产生喂狗指令 wdr。

AVR libc 库函数索引

原文作者为 Enno Luebbers (LUEBBERS @ USERS.SOURCEFORGE.NET)，适用版本为 2001/07/27。

一．头文件介绍：

ctype.h：字符类型函数

eeprom.h：EEPROM 访问函数

errno.h：错误处理

ina90.h：与 IAR C 兼容的头文件

interrupt.h：中断处理

inttypes.h：定义不同的数据类型

io-avr.h：包含了正确的 ioXXX.h 头文件

io.h：包含了其他的 I/O 头文件

ioXXX.h：为不同的 AVR 器件定义了 I/O

iomacros.h：访问 I/O 的宏

math.h：数学函数

pgmspace.h：与 IAR C 兼容的头文件

progmem.h：与 pgmspace.h 相同

setjmp.h：远跳转函数

sig-avr.h：中断处理

signal.h：与 sig-avr.h 相同。建议不使用

stdlib.h：杂项

string-avr.h：字符串操作函数

string.h：更多的字符串操作函数

timer.h：定时器控制函数

twi.h：I^2C 函数

wdt.h：看门狗定时器控制函数

二．按字母排列的函数列表：

void abort();

double acos(double x);

double asin(double x);

double atan(double x);

long atoi(char *p);

long atol(char *p);

bit_is_clear(port, bit);

bit_is_set(port, bit);

void bsearch(const void *key, const void *base, size_t nmemb);

BV(x);

cbi(port, bit);

double ceil(doublce x);

cli();

```
double cos( double x );
double cosh( double x );
div_t div( int x, int y );
int eeprom_is_ready();
unsigned char eeprom_rb(unsigned int addr);
void eeprom_read_block(void *buf, unsigned int addr, size_t n);
unsigned int eeprom_rw(unsigned int addr);
void eeprom_wb(unsigned int addr, unsigned char val);
void enable_external_int( unsigned char ints );
int errno();
double exp( double x );
double fabs( double x );
double floor( double x );
double fmod( double x, double y );
void free( void *ptr );
double frexp( double x, int *exp );
inp( port );
INTERRUPT( signame )
double inverse( double x );
int isalnum(int __c);
int isalpha(int __c);
int isascii(int __c);
int iscntrl(int __c);
```

```
int isdigit(int __c);

int isgraph(int __c);

int islower(int __c);

int isprint(int __c);

int ispunct(int __c);

int isspace(int __c);

int isupper(int __c);

int isxdigit(int __c);

char itoa( int value, char *string, int radix )

long labs( long x );

double ldexp( double x, int exp );

ldiv_t ldiv( lomg x, long y );

double log( double x );

double log10( double x );

void longjmp( jmp_buf env, int val );

loop_until_bit_is_clear( port, bit );

loop_until_bit_ist_set( port, bit );

void malloc( size_t size );

void memchr( void *s, char c, size_t n );

int memcmp( const void *s1, const void *s2, size_t n );

void memcpy( void *to, void *from, size_t n );

void memmove( void *to, void *from, size_t n );

void memset( void *s, int c, size_t n );
```

double modf(double x, double *iptr);

outp(value, port);

parity_even_bit(val);

double pow(double x, double y);

void qsort(void *base, size_t nmemb, size_t size, __compar_fn_t compar);

sbi(port, bit);

sei();

int setjmp(jmp_buf env);

SIG_ADC()

SIG_COMPARATOR()

SIG_EEPROM()

SIG_INPUT_CAPTURE1()

SIG_INTERRUPT0 through SIG_INTERRUPT7()

SIG_OUTPUT_COMPARE0()

SIG_OUTPUT_COMPARE1A()

SIG_OUTPUT_COMPARE2()

SIG_OVERFLOW0()

SIG_OVERFLOW1()

SIG_OVERFLOW2()

SIG_SPI()

SIG_UART1_DATA()

SIG_UART1_RECV()

SIG_UART1_TRANS()

```
SIG_UART_DATA()

SIG_UART_RECV()

SIG_UART_TRANS()

SIGNAL( signame );

double sin( double x );

double sinh( double x );

double sqrt( double x );

double square( double x );

extern int strcasecmp(const char *s1, const char *s2);

char strcat( char *dest, char *src );

char strchr( const char *s, int c );

int strcmp( const char *s1, const char* s2 );

char strcpy( char *dest, char *src );

strdupa( s );

size_t strlen( char *s );

extern char strlwr(char *);

extern int strncasecmp(const char *, const char *, size_t);

char strncat( char *dest, char *src, size_t n );

int strncmp( const char *s1, const char* s2, size_t n );

char strncpy( char *dest, char *src, size_t n );

strndupy( s, n );

size_t strnlen( const char *s, size_t maxlen );

char strrchr( const char *s, int c );
```

extern char strrev(char *s1);

extern char strstr(const char *haystack, const char *needle);

double strtod(char *, char **);

double strtod(const char *s, char **endptr);

long strtol(const char *nptr, char **endptr, int base);

unsigned long strtoul(const char *nptr, char **endptr, int base);

extern char strupr(char *);

double tan(double x);

double tanh(double x);

void timer0_source(unsigned int src);

coid timer0_start();

void timer0_stop();

void timer_enable_int(unsigned char ints);

int toascii(int __c);

int tolower(int __c);

int toupper(int __c);

wdt_disable();

wdt_enable(timeout);

wdt_reset();

三．按头文件介绍：

1．CTYPE.H

int isalnum(int __c); 如果 c 为字母或数字则返回 1，否则返回 0

int isalpha(int __c); 如果 c 为字母则返回 1，否则返回 0

int isascii(int __c); 如果 c 为 ASCII 码则返回 1，否则返回 0

int iscntrl(int __c); 如果 c 为控制字符则返回 1，否则返回 0

int isdigit(int __c); 如果 c 为数字则返回 1，否则返回 0

int isgraph(int __c); 如果 c 为可打印字符（不包括空格）则返回 1，否则返回 0

int islower(int __c); 如果 c 为小写字母则返回 1，否则返回 0

int isprint(int __c); 如果 c 为可打印字符（包括空格）则返回 1，否则返回 0

int ispunct(int __c); 如果 c 为标点符号则返回 1，否则返回 0

int isspace(int __c); 如果 c 为空格、'\n'、 '\f'、 '\r'、 '\t'、 '\v'之一则返回 1，否则返回

int isupper(int __c); 如果 c 为大写字母则返回 1，否则返回 0

int isxdigit(int __c); 如果 c 为 16 进制数则返回 1，否则返回 0

int toascii(int __c); 将 c 转换为 7bit ASCII 字符

int tolower(int __c); 将 c 转换为小写

int toupper(int __c); 将 c 转换为大写

2．EEPROM.H

int eeprom_is_ready() /* 宏 */ 若寄存器 EECR 的 EEWE 位为 0 则返回非零值

unsigned char eeprom_rb(unsigned int addr); 从 EEPROM 地址为'addr'处读出一个字节

unsigned int eeprom_rw(unsigned int addr); 从 EEPROM 处读出 2 个字节，低地址位于低地址

void eeprom_wb(unsigned int addr, unsigned char val); 将数据写入 EEPROM，地

址为'addr'

void eeprom_read_block(void *buf, unsigned int addr, size_t n); 从 EEPROM 地址'addr'处读出 n 个数据到'buf'

_EEPUT(addr, val) eeprom_wb(addr, val)

_EEGET(var, addr) (var) = eeprom_rb(addr) IAR C 的兼容函数

3．ERRNO.H

int errno; 系统定义的错误代码

4．INA90.H

用于将 IAR C 的应用移植到 AVR-GCC

5．INTERRUPT.H

sei(); 使能中断，是宏

cli(); 禁止中断，是宏

void enable_external_int(unsigned char ints); 将'ints'赋值给 EIMSK/GIMSK

void timer_enable_int(unsigned char ints); 将 'ints'赋值给 TIMSK

6．INTTYPES.H

typedef signed char int8_t;

typedef unsigned char uint8_t;

typedef int int16_t;

typedef unsigned int uint16_t;

typedef long int32_t;

typedef unsigned long uint32_t;

typedef long long int64_t;

typedef unsigned long long uint64_t;

typedef int16_t intptr_t;

typedef uint16_t uintptr_t;

注意：编译器的 -mint8 选项将 int 定义为 8 位数值

7．IO-AVR.H

自动为目标处理器包含头文件 ioXXX.h

8．IO.H

包含 both io-avr.h 和 iomacros.h.

9．IOXXX.H

I/O 寄存器定义

10．IOMACROS.H

BV(x); 完成(1 << x)功能，是一个宏

inp(port); 从端口'port'读入一个字节。可以自动识别 I/O 或存储器地址，并相应地使用宏__inb 或__mmio

outp(value, port); 将'value'写入端口'port'

cbi(port, bit); 清零端口'port'的位'bit'

sbi(port, bit); 置位

bit_is_set(port, bit); 若端口'port'的位'bit'为 1 则返回非零值

bit_is_clear(port, bit); 若端口'port'的位'bit'为 0 则返回非零值

loop_until_bit_ist_set(port, bit); 循环至端口'port'的位'bit'置位

loop_until_bit_is_clear(port, bit); 循环至端口'port'的位'bit'清零

parity_even_bit(val); ？

11．MATH.H

常数π：

M_PI = 3.14159265358979323846264 3

2 的平方根：

M_SQRT2 = 1.41421356237309504880168 87

double cos(double x);

double fabs(double x); x 的绝对值

double fmod(double x, double y); 返回 x/y 的余数

double modf(double x, double *iptr); 返回 x 的小数部分，整数赋值给*iptr.

double sin(double x);

double sqrt(double x); 返回 x 的平方根

double tan(double x);

double floor(double x); 返回最接近 x 而又小于它的整数

double ceil(doublce x); 返回最接近 x 而又大于它的整数

double frexp(double x, int *exp); 将 x 归一化。小数部分为返回值，指数保存在 *exp

double ldexp(double x, int exp); 返回 x*2^exp

double exp(double x); 返回 e^x

double cosh(double x);

double sinh(double x);

double tanh(double x);

double acos(double x);

double asin(double x);

double atan(double x);

double log(double x);

double log10(double x);

double pow(double x, double y); 返回 x^y.

double strtod(const char *s, char **endptr); 将 ASCII 字符串转换为 double 数

double square(double x); 返回 x^2;

double inverse(double x); 返回 1/x;

12．PGMSPACE.H

13．PROGMEM.H

包含 pgmspace.h.

14．SETJMP.H

int setjmp(jmp_buf env); 声明一个长跳转目标以指示 longjmp 跳转到此处。

void longjmp(jmp_buf env, int val); 执行长跳转到 setjmp 定义的'env' 处, setjmp

返回'val'

15．SIG-AVR.H

定义中断向量符号。

SIG_INTERRUPT0 ~ SIG_INTERRUPT7：外部中断 0~7 的中断例程名称。不是所有的 AVR 器件都具有这些中断，请参见数据手册。

SIG_OUTPUT_COMPARE2

SIG_OVERFLOW2

SIG_INPUT_CAPTURE1

SIG_OUTPUT_COMPARE1A

SIG_OVERFLOW1

SIG_OUTPUT_COMPARE0

SIG_OVERFLOW0

SIG_SPI

SIG_UART_RECV：UART0 接收完成中断

SIG_UART1_RECV

SIG_UART_DATA：UART 数据寄存器空中断

SIG_UART1_DATA

SIG_UART_TRANS：UART0 发送结束中断

SIG_UART1_TRANS

SIG_ADC

SIG_EEPROM

SIG_COMPARATOR

SIGNAL(signame); 为信号'signame'定义中断例程。例程中全局中断使能位一直为 0

INTERRUPT(signame); 为信号'signame'定义中断例程。例程中全局中断使能位置位，以实现中断嵌套

16．SIGNAL.H

包含了 sig-avr.h。建议不要使用

17．STDLIB.H

typedef struct {

int quot;

int rem;

} div_t;

typedef struct {

long quot;

long rem;

} ldiv_t;

typedef int (*__compar_fn_t)(const void *, const void *); 用于比较函数，如 qsort().

void abort();

long labs(long x); 返回 x 的绝对值

void *bsearch(const void *key, const void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *));

div_t div(int x, int y); x/y

ldiv_t ldiv(lomg x, long y);

void qsort(void *base, size_t nmemb, size_t size, __compar_fn_t compar); long strtol(const char *nptr, char **endptr, int base);

unsigned long strtoul(const char *nptr, char **endptr, int base);

long atol(char *p); 将字符串'p'转换为长整型

long atoi(char *p); 将字符串'p'转换为长整数

void *malloc(size_t size);

void free(void *ptr);

double strtod(char *, char **);

char *itoa(int value, char *string, int radix);

以下函数还未实现：

atexit(), atof(), calloc(), rand(), realloc(), srand();

17．STRING-AVR.H

void *memcpy(void *to, void *from, size_t n);

void *memmove(void *to, void *from, size_t n);

void *memset(void *s, int c, size_t n);

int memcmp(const void *s1, const void *s2, size_t n);

void *memchr(void *s, char c, size_t n);

size_t strlen(char *s);

char *strcpy(char *dest, char *src);

char *strncpy(char *dest, char *src, size_t n);

char *strcat(char *dest, char *src);

char *strncat(char *dest, char *src, size_t n);

int strcmp(const char *s1, const char* s2);

int strncmp(const char *s1, const char* s2, size_t n);

strdupa(s); 复制's'

strndupa(s, n); 返回's'的头'n'个字节

char *strchr(const char *s, int c); 返回's'里第一个'c'的位置

char *strrchr(const char *s, int c); 返回's'里最后一个'c'的位置

size_t strnlen(const char *s, size_t maxlen);

18．STRING.H

extern void *memccpy(void *dest, const void *src, int c, size_t n); 从'src'拷贝至多’n’个字节到'dest'，直至找到字符'c'

extern void *memchr(const void *, int, size_t); 参见 string-avr.h.

extern int memcmp(const void *, const void *, size_t); 参见 string-avr.h.

extern void *memcpy(void *, const void *, size_t); 参见 string-avr.h.

extern void *memmove(void *dest, const void *src, size_t n); 参见 string-avr.h.

extern void *memset(void *, int, size_t); 参见 string-avr.h.

extern char *strcat(char *, const char *); 参见 string-avr.h.

extern char *strchr(const char *, int); 参见 string-avr.h.

extern int strcmp(const char *, const char *); 参见 string-avr.h.

extern char *strcpy(char *, const char *); 参见 string-avr.h.

extern int strcasecmp(const char *s1, const char *s2); 比较's1'和's2', 不考虑大小写

extern size_t strlen(const char *); 参见 string-avr.h.

extern char *strlwr(char *); ？

extern char *strncat(char *, const char *, size_t); 参见 string-avr.h.

extern int strncmp(const char *, const char *, size_t); 参见 string-avr.h.

extern char *strncpy(char *, const char *, size_t); 参见 string-avr.h.

extern int strncasecmp(const char *, const char *, size_t); 比较's1' 和 's2'的'n' 个字节不考虑大小写

extern size_t strnlen(const char *, size_t); 参见 string-avr.h.

extern char *strrchr(const char *, int); 参见 string-avr.h.

extern char *strrev(char *s1); ？

extern char *strstr(const char *haystack, const char *needle); 在'haystack'中查找'needle'并返回地址

extern char *strupr(char *);.

19．TIMER.H

enum {

STOP = 0,

CK = 1,

CK8 = 2,

CK64 = 3,

CK256 = 4,

CK1024 = 5,

T0_FALLING_EDGE = 6,

T0_RISING_EDGE = 7

};

void timer0_source(unsigned int src); 将'src'赋值给寄存器 TCCR0

void timer0_stop(); 通过对寄存器 TCNT0 清零来停止定时器 0

coid timer0_start();通过对寄存器 TCNT0 写入 1 来起动定时器 0

20．TWI.H

21．WDT.H

wdt_reset(); 喂狗

wdt_enable(timeout); 使能看门狗，并设置溢出时间为'timeout'

wdt_disable(); 关闭看门狗

附录：AVR-GCC 配置

汇编选项

选项	描述
-mmcu=*name*	指定目标器件 *name* 可以为：at90s1200，at90s2313，at90s2323，at90s2333，at90s2343， at904433， at90s8515， at90s8535， atmega103，atmega161

注：*name* 还可以为 avr1（适用于 TINY 和 1200），avr2，avr3，avr4，avr5。其

中 avr1，avr2，avr3 不支持增强型 AVR 器件（Enhanced core），而 avr4，avr5 支持；avr2，avr4 支持 FLASH 不大于 8K 的器件，而 avr3，avr5 支持大于 8K 的器件，如 16K，32K，128K，以及 AT94K。增强型 AVR 指的是至少支持乘法的器件。

寄存器使用

如果用户需要进行汇编与 C 的混合编程，必须了解寄存器的使用。

1．寄存器使用

r0　可用做暂时寄存器。如果用户汇编代码使用了 r0，且要调用 C 代码，则在调用之前必须保存 r0。C 中断例程会自动保存和恢复 r0。

r1　C 编译器假定此寄存器内容为“0”。如果用户使用了此寄存器，则在汇编代码返回之前须将其清零。C 中断例程会自动保存和恢复 r1。

r2-r17，r28，r29　C 编译器使用这些寄存器。如果用户汇编代码需要使用这些寄存器，则必须保存并恢复这些寄存器。

R18-r27，r30，r31　如果用户汇编代码不调用 C 代码则无需保存和恢复这些寄存器。如果用户要调用 C 代码，则在调用之前须保存。

2．函数调用规则

参数表：函数的参数由左至右分别分配给 r25 到 r8。每个参数占据偶数个寄存器。若参数太多以至 r25 到 r8 无法容纳，则多出来的参数将放入堆栈。

返回值：8 位返回值存放在 r24。16 位返回值存放在 r25:r24。32 位返回值存放在 r25:r24:r23:r22。64 位返回值存放在 r25:r24:r23:r22:r21:r20:r19:r18。